# µPuppet: A Declarative Subset of the Puppet Configuration Language

# $\mu$Puppet: A Declarative Subset of the Puppet Configuration Language

**Weili Fu[1], Roly Perera[1,2], Paul Anderson[1], and James Cheney[1]**

1 **School of Informatics, University of Edinburgh, Edinburgh, UK**
`weili.fu@ed.ac.uk`, `roly.perera@ed.ac.uk`, `dcspaul@ed.ac.uk`, `jcheney@inf.ed.ac.uk`
2 **School of Computing Science, University of Glasgow, Glasgow, UK**
`roly.perera@glasgow.ac.uk`

## Abstract

Puppet is a popular declarative framework for specifying and managing complex system configurations. The Puppet framework includes a domain-specific language with several advanced features inspired by object-oriented programming, including user-defined resource types, 'classes' with a form of inheritance, and dependency management. Like most real-world languages, the language has evolved in an ad hoc fashion, resulting in a design with numerous features, some of which are complex, hard to understand, and difficult to use correctly.

We present an operational semantics for $\mu$Puppet, a representative subset of the Puppet language that covers the distinctive features of Puppet, while excluding features that are either deprecated or work-in-progress. Formalising the semantics sheds light on difficult parts of the language, identifies opportunities for future improvements, and provides a foundation for future analysis or debugging techniques, such as static typechecking or provenance tracking. Our semantics leads straightforwardly to a reference implementation in Haskell. We also discuss some of Puppet's idiosyncrasies, particularly its handling of classes and scope, and present an initial corpus of test cases supported by our formal semantics.

**1998 ACM Subject Classification** D.2.9 [Software Engineering] Management – Software configuration management

**Keywords and phrases** configuration languages; Puppet; operational semantics

**Digital Object Identifier** 10.4230/LIPIcs.ECOOP.2017.9

## 1 Introduction

Managing a large-scale data center consisting of hundreds or thousands of machines is a major challenge. Manual installation and configuration is simply impractical, given that each machine hosts numerous software components, such as databases, web servers, and middleware. Hand-coded configuration scripts are difficult to manage and debug when multiple target configurations are needed. Moreover, misconfigurations can potentially affect millions of users. Recent empirical studies [22, 11] attribute a significant proportion of system failures to misconfiguration rather than bugs in the software itself. Thus better support for specifying, debugging and verifying software configurations is essential to future improvements in reliability [21].

A variety of *configuration frameworks* have been developed to increase the level of automation and reliability. All lie somewhere on the spectrum between "imperative" and "declarative". At the imperative end, developers use conventional scripting languages to automate common tasks. It is left to the developer to make sure that steps are performed


in the right order, and that any unnecessary tasks are not (potentially harmfully) executed anyway. At the declarative end of the spectrum, the desired system configuration is *specified* in some higher-level way and it is up to the configuration framework to determine how to *realise* the specification: that is, how to generate a compliant configuration, or adapt an already-configured system to match a new desired specification.

Most existing frameworks have both imperative and declarative aspects. Chef [13], CFEngine [23], and Ansible [8] are imperative in relation to dependency management; the order in which tasks are run must be specified. Chef and CFEngine are declarative in that a configuration is specified as a desired target state, and only the actions necessary to end up in a compliant state are executed. (This is called *convergence* in configuration management speak.) The Puppet framework [18] lies more towards the declarative end, in that the order in which configuration tasks are carried out is also left mostly to the framework. Puppet also provides a self-contained *configuration language* in which specifications are written, in contrast to some other systems. (Chef specifications are written in Ruby, for example, whereas Ansible is YAML-based.)

Configuration languages often have features in common with general-purpose programming languages, such as variables, expressions, assignment, and conditionals. Some, including Puppet, also include "object-oriented" features such as classes and inheritance. However, (declarative) configuration languages differ from regular programming or scripting languages in that they mainly provide mechanisms for specifying, rather than realising, configurations. While some "imperative" features that can directly mutate system state are available in Puppet, their use is generally discouraged.

Like most real-world languages, configuration languages have largely evolved in an ad hoc fashion, with little attention paid to their semantics. Given their infrastructural significance, this makes them an important (although challenging) target for formal study: a formal model can clarify difficult or counterintuitive aspects of the language, identify opportunities for improvements and bug-fixes, and provide a foundation for static or dynamic analysis techniques, such as typechecking, provenance tracking and execution monitoring. In this paper, we investigate the semantics of the configuration language used by the Puppet framework. Puppet is a natural choice because of its DSL-based approach, and the fact that it has seen widespread adoption. The 2016 PuppetConf conference attracted over 1700 Puppet users and developers and sponsorship from over 30 companies, including Cisco, Dell, Microsoft, Google, Amazon, RedHat, VMWare, and Citrix.

An additional challenge for the formalisation of real-world languages is that they tend to be moving targets. For example, Puppet 4.0, released in March 2015, introduced several changes that are not backwards-compatible with Puppet 3, along with a number of non-trivial new features. In this paper, we take Puppet 4.8 (the version included with Puppet Enterprise 2016.5) as the baseline version of the language, and define a subset called $\mu$Puppet (pronounced "muppet") that includes the established features of the language that appear most important and distinctive; in particular, it includes the constructs `node`, `class`, and `define`. These are used in almost all Puppet programs (called *manifests*). We chose to exclude some features that are either deprecated or not yet in widespread use, or whose formalisation would add complication without being particularly enlightening, such as regular expressions and string interpolation.

The main contributions of this paper are:

1. a formalisation of $\mu$Puppet, a subset of Puppet 4.8;
2. a discussion of simple metatheoretic properties of $\mu$Puppet such as determinism, monotonicity and (non-)termination;

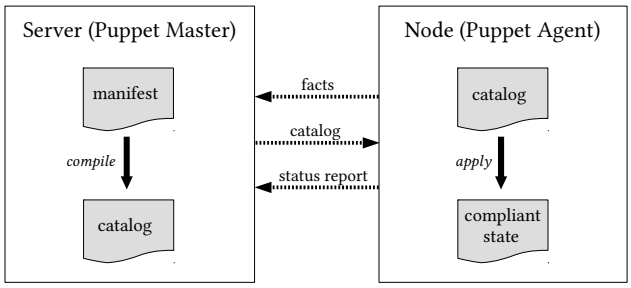


**Figure 1** Puppet overview

3. a reference implementation of $\mu$Puppet in Haskell;
4. a corpus of test cases accepted by our implementation;
5. a discussion of the more complex features not handled by $\mu$Puppet.

We first give an overview of the language via some examples (Section 2), covering some of the more counterintuitive and surprising parts of the language. Next we define the abstract syntax and a small-step operational semantics of $\mu$Puppet (Section 3). We believe ours to be the first formal semantics a representative subset of Puppet; although recent work by Shambaugh et al. [17] handles some features of Puppet, they focus on analysis of the "realisation" phase and do not present a semantics for the `node` or `class` constructs or for inheritance (although their implementation does handle some of these features). We use a small-step operational semantics (as opposed to large-step or denotational semantics) because it is better suited to modelling some of the idiosyncratic aspects of Puppet, particularly the sensitivity of scoping to evaluation order. We focus on unusual or novel aspects of the language in the main body of the paper; the full set of rules are given in the appendix of the extended paper [7]. Section 4 discusses some properties of $\mu$Puppet, such as determinism and monotonicity, that justify calling it a 'declarative' subset of Puppet. Section 5 describes our implementation and how we validated our rules against the actual behaviour of Puppet, and discusses some of the omitted features. Sections 6 and 7 discuss related work and present our conclusions.

## 2 Overview of Puppet

Puppet uses several terms – especially *compile*, *declare*, and *class* – in ways that differ from standard usage in programming languages and semantics. We introduce these terms with their Puppet meanings in this section, and use those meanings for the rest of the paper. To aid the reader, we include a glossary of Puppet terms in the appendix of the extended paper [7].

The basic workflow for configuring a single machine (*node*) using Puppet is shown in Figure 1. A *Puppet agent* running on the node to be configured contacts the *Puppet master* running on a server, and sends a check-in request containing local information, technically called *facts*, such as the name of the operating system running on the client node. Using this information, along with a centrally maintained configuration specification called the *manifest*, the Puppet master *compiles* a *catalog* specific to that node. The manifest is written in a high-level language, the Puppet programming language (often referred to simply as Puppet), and consists of *declarations* of *resources*, along with other program constructs used to define resources and specify how they are assigned to nodes. A resource is simply a collection of key-value pairs, along with a *title*, of a particular *resource type*; "declaring" a resource

means specifying that a resource of that type exists in the target configuration. The catalog resulting from compilation is the set of resources computed for the target node, along with other metadata such as ordering information among resources. The Puppet master may fail to compile a manifest due to compilation errors. In this case, it will not produce a compiled catalog. If compilation succeeds, the agent receives the compiled catalog and *applies* it to reconfigure the client machine, ideally producing a compliant state. Puppet separates the compilation of manifests and the deployment of catalogs. After deploying the catalog, either the changed configuration meets the desired configuration or there are some errors in it that cause system failures. Finally, the agent sends a status report back to the master indicating success or failure.

Figure 1 depicts the interaction between a single agent and master. In a large-scale system, there may be hundreds or thousands of nodes configured by a single master. The manifest can describe how to configure all of the machines in the system, and parameters that need to be coordinated among machines can be specified in one place. A given run of the Puppet manifest compiler considers only a single node at a time.

## 2.1 Puppet: key concepts

We now introduce the basic concepts of the Puppet language – manifests, catalogs, resources, and classes – with reference to various examples. We also discuss some behaviours which may seem surprising or unintuitive; clarifying such issues is one reason for pursuing a formal definition of the language. The full Puppet 4.8 language has many more features than presented here. A complete list of features and the subset supported by $\mu$Puppet are given in the appendix of the extended paper [7].

### 2.1.1 Manifests and catalogs

Figure 2 shows a typical manifest, consisting of a *node definition* and various *classes* declaring resources, which will be explained in § 2.1.4 below. Node definitions, such as the one starting on line 1, specify how a single machine or group of machines should be configured. Single machines can be specified by giving a single hostname, and groups of machines by giving a list of hostnames, a regular expression, or `default` (as in this example). The `default` node definition is used if no other definition applies.

In this case the only node definition is `default`, and so compiling this manifest for any node results in the catalog on the right of Figure 2. In this case the catalog is a set of resources of type `file` with titles `config1`, `config2` and `config3`, each with a collection of attribute-value pairs. Puppet supports several persistence formats for catalogs, including YAML; here we present the catalog using an abstract syntax which is essentially a sub-language of the language of manifests. The `file` resource type is one of Puppet's many built-in resource types, which include other common configuration management concepts such as `user`, `service` and `package`.

### 2.1.2 Resource declarations

Line 11 of the manifest in Figure 2 shows how the `config1` resource in the catalog was originally declared. The `path` attribute was specified explicitly as a string literal; the other attributes were given as variable references of the form $\$x$. Since a resource with a given title and type is global to the entire catalog, it may be declared only once during a given compilation. A *compilation error* results if a given resource is declared more than once. Note

```
1  node default {
2    $source = "/source"
3    include service1
4  }
5
6  class service1 {
7    $mode = 123
8
9    include service2
10
11   file { "config1":
12     path => "path1",
13     source => $source,
14     mode => $mode,
15     checksum => $checksum,
16     provider => $provider,
17     recurse => $recurse
18   }
19
20   $checksum = "md5"
21 }
22
23 class service2 inherits service3 {
24   $recurse = true
25
26   file { "config2":
27     path => "path2",
28     source => $source,
29     mode => $mode,
30     checksum => $checksum,
31     provider => $provider,
32     recurse => $recurse
33   }
34 }
35
36 class service3 {
37   $provider = posix
38
39   file { "config3":
40     path => "path3",
41     mode => $mode,
42     checksum => $checksum,
43     recurse => $recurse
44   }
45 }
```

```
1  file { "config3":
2      path => "path3"
3  }
4  file { "config2":
5      path => "path2",
6      source => "/source",
7      provider => "posix",
8      recurse => true
9  }
10 file { "config1":
11     path => "path1",
12     source => "/source",
13     mode => 123
14 }
```

**Figure 2** Example manifest (left); compiled catalog (right)

that what Puppet calls a "compilation error" is a purely dynamic condition, and so is really a runtime error in conventional terms.

The ordering of attributes within a resource is not significant; by default they appear in the catalog in the order in which they were declared. Optionally they can be sorted (by specifying ordering constraints) or randomised. Sorting is usually recommended over relying on declaration order [16].

### 2.1.3 Variables and strict mode

Puppet lacks variable declarations in the usual sense; instead variables are implicitly declared when they are assigned to. A compilation error results if a given variable is assigned to more than once in the same scope. As we saw above, unqualified variables, whether being read or assigned to, are written in "scripting language" style $\$x$.

Puppet allows variables to be used before they are assigned, in which case their value is a special "undefined" value `undef`, analogous to Ruby's `nil` or JavaScript's `undefined`. By default, attributes only appear in the compiled output if their values are defined. Consider the variables `$mode` and `$checksum` introduced by the assignments at lines 7 and 20 in the

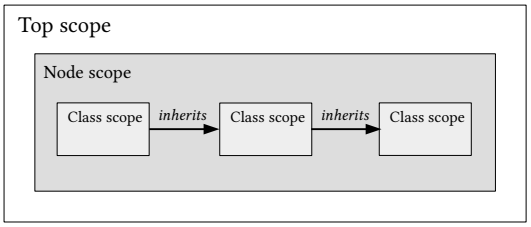


**Figure 3** Two aspects of scope: parent scopes (shown as containment), and inheritance chains

manifest in Figure 2. The ordering of these variables relative to the file resource `config1` is significant, because it affects whether they are in scope. Since `$mode` is defined *before* `config1`, its value can be read and assigned to the attribute `mode`. In the compiled catalog, `mode` thus appears as an attribute of `config1`. On the other hand `$checksum` is assigned *after* `config1`, and is therefore undefined when read by the code which initialises the `checksum` attribute. Thus `checksum` is omitted from the compiled version of `config1`.

Since relying on the values of undefined variables is often considered poor practice, Puppet provides a *strict* mode which treats the use of undefined variables as an error. For similar reasons, and also to keep the formal model simple, $\mu$Puppet always operates in strict mode. We discuss the possibility of relaxing this in Section 5.3.

### 2.1.4 Classes and includes

Resource declarations may be grouped into *classes*. However, Puppet "classes" are quite different from the usual concept of classes in object-oriented programming – they define collections of resources which can be declared together by *including* the class. This is sometimes called *declaring* the class, although there is a subtle but important distinction between "declaring" and "including" which we will return to shortly.

In Figure 2, it is the inclusion into the node definition of class `service1` which explains the appearance of `config1` in the catalog, and in turn the inclusion into `service1` of class `service2` which explains the appearance of `config2`. (The fact that `config3` also appears in the output relates to inheritance, and is discussed in §2.1.6 below.) Inclusion is idempotent: the same class may be included multiple times, but doing so only generates a single copy of the resources in the catalog. This allows a set of resources to be included into all locations in the manifest which depend on them, without causing errors due to duplicate declarations of the same resource.

To a first approximation, including a class into another class obeys a lexical scope discipline, meaning names in the including class are not visible in the included class. However inclusion into a node definition has a quite different behaviour: it introduces a containment relation between the node definition and the class, meaning that names scoped to the node definition are visible in the body of the included class. Thus in Figure 2, although the variable `$mode` defined in `service1` is not in scope inside the included class `service2` (as per lexical scoping), the `$source` variable defined in the node definition *is* in scope in `service1`, because `service1` is included into the node scope.

This is similar to the situation in Java where a class asserts its membership of a package using a package declaration, except here the node definition pulls *in* the classes it requires. The subtlety is that it is actually when a class is *declared* (included for the first time, dynamically speaking) that any names in the body of the class are resolved. If the *usage* of a class happens to change so that it ends up being declared in so-called *top* scope (the root

namespace usually determine at check-in time), it may pick up a different set of bindings. Thus including a class, although idempotent, has a "side effect" – binding the names in the class – making Puppet programs potentially fragile. More of the details of scoping are given in the language reference manual [1].

### 2.1.5 Qualified names

A definition which is not in scope can be accessed using a *qualified* name, using a syntax reminiscent of C++ and Java, with atomic names separated by the token `::`. For example, in Figure 4 above, `$::osfamily` refers to a variable in the top scope, while `$::ssh::params::sshd_package` is an absolute reference to the `$sshd_package` variable of class `ssh::params`.

Less conventionally, Puppet also allows the name of a class to be a qualified name, such as `ssh::params` in Figure 4. Despite the suggestive syntax, which resembles a C++ member declaration, this is mostly just a convention used to indicate related classes. In particular, qualified names used in this way do not require any of the qualifying prefixes to denote an actual namespace. (Although see the discussion in Section 5.3 for an interaction between this feature and nested classes, which $\mu$Puppet does not support.)

### 2.1.6 Inheritance and class parameters

Classes may *inherit* from other classes; the inheriting class inherits the variables of the parent class, including their values. In the earlier example (Figure 2), `service2` inherits the value of `$provider` from `service3`. Including a derived class implicitly includes the inherited class, potentially causing the inherited class to be declared (in the Puppet sense of the word) when the derived class is declared:

> *When you declare a derived class whose base class hasn't already been declared, the base class is immediately declared in the current scope, and its parent assigned accordingly. This effectively "inserts" the base class between the derived class and the current scope. (If the base class has already been declared elsewhere, its existing parent scope is not changed.)*

This explains why `config3` appears in the compiled catalog for Figure 2.

Since the scope in which a class is eventually declared determines the meaning of the names in the class (§ 2.1.4 above), inheritance may have surprising (and non-local) consequences. At any rate, the use of inheritance for most use cases is now discouraged.[1] The main exception is the use of inheritance to specify default values; this is the scenario illustrated in Figure 4.

Line 1 of Figure 4 introduces class `ssh::params`, which assigns to variable `$sshd_package` a value conditional on the operating system name `$::osfamily` (line 2). The class `ssh` (line 8) inherits from `ssh::params`. It also defines a *class parameter* `$ssh_pkg` (before the `inherits` clause), and uses the value of the `$sshd_package` variable in the inherited class as the default value for the parameter. Because an inherited class is processed before a derived class, the value of `$sshd_package` is available at this point.

The value of the parameter `$ssh_pkg` is then used as the title of the `package` resource declared in the `ssh` class (line 9) specifying that the relevant software package exists in the target configuration. The last construct is a node definition specifying how to configure the

[1] `https://docs.puppet.com/puppet/latest/style_guide.html`, section 11.1.

```
1  class ssh::params {
2    case $::osfamily {
3    "Debian": { $sshd_package  = "ssh" }
4    "RedHat": { $sshd_package  = "openssh-server" }
5    default:  { fail("SSH class not supported") }
6    }
7  }
8  class ssh ($ssh_pkg = $::ssh::params::sshd_package) inherits ssh::params {
9    package { $ssh_pkg:
10     ensure => installed
11   }
12 }
13 node "ssh.example.com" {
14   include ssh
15 }
```

**Figure 4** Example manifest showing recommended use of inheritance for setting default parameters

machine with hostname `ssh.example.com`. If host `ssh.example.com` is a Debian machine, the result of compiling this manifest is a catalog containing the following `package` resource:

```
1  package { "ssh" :  ensure => installed }
```

### 2.1.7 Class statements

Figure 5 defines a class `c` with three parameters. The `class` statement (line 31) can be used to include a class and provide values for (some of) the parameters. In the resulting catalog, the `from_class` resource has `backup` set to `true` (from the explicit argument), `mode` set to `123` (because no `mode` argument is specified), and `source` set to `'/default'` (because the `path` variable is undefined at the point where the class is declared (line 31)).

However, the potential for conflicting parameter values means that multiple declarations with parameters are not permitted, and the `class` statement must be used instead (which only allows a single declaration).

### 2.1.8 Defined resource types

*Defined resource types* are similarly to classes, but provide a more flexible way of introducing a user-defined set of resources. Definition `d` (line 14) in Figure 5 introduces a defined resource type. The definition looks very similar to a class definition, but the body is a macro which can be instantiated (line 36) multiple times with different parameters.

Interestingly, the `path` attribute in the `from_class` file is undefined in the result, apparently because the assignment `$path = '/path'` follows the declaration of the class — however, in the `from_define` file, `path` *is* defined as `'/path'`! The reason appears to be that defined resources are added to the catalog and re-processed after other manifest constructs.[2]

## 3 $\mu$Puppet

We now formalise $\mu$Puppet, a language which captures many of the essential features of Puppet. Our goal is not to model all of Puppet's idiosyncrasies, but instead to attempt to capture the 'declarative' core of Puppet, as a starting point for future study. As we discuss later, Puppet also contains several non-declarative features whose behaviour can be

[2] `http://puppet-on-the-edge.blogspot.co.uk/2014/04/getting-your-puppet-ducks-in-row.html`

```
class c (
  $backupArg = false,
  $pathArg = "/default",
  $modeArg = 123 ) {

  file { "from_class":
    backup => $backupArg,
    source => $pathArg,
    path => $path,
    mode => $modeArg
  }
}

define d (
  $backupArg = false,
  $pathArg = "/default",
  $modeArg = 123 ) {

  file { "from_define":
    backup => $backupArg,
    source => $pathArg,
    path => $path,
    mode => $modeArg
  }
}

node default {

  $backup = true

  class { c:
    backupArg => $backup,
    pathArg => $path
  }

  d { "service3":
    backupArg => $backup,
    pathArg => $path
  }

  $path = "/path"
}
```

```
file { "from_class":
    backup => true,
    source => "/default",
    mode => 123
}
file { "from_define":
    path => "/path",
    backup => true,
    source => "/default",
    mode => 123
}
```

**Figure 5** Manifest with class parameters and defined resource types (left); catalog (right)

counterintuitive and surprising; their use tends to be discouraged in Puppet's documentation and by other authors [16].

## 3.1 Abstract syntax

The syntax of $\mu$Puppet manifests $m$ is defined in Figure 6, including expressions $e$ and statements $s$. Constant expressions in $\mu$Puppet can be integer literals $i$, string literals $w$, or boolean literals `true` or `false`. Other expressions include arithmetic and boolean operations, variable forms $\$x$, $\$::x$ and $\$::a::x$. Here, $x$ stands for variable names and $a$ stands for class names. *Selectors* $e$ ? $\{M\}$ are conditional expressions that evaluate $e$ and then conditionally evaluate the first matching branch in $M$. Arrays are written $[e_1, \ldots, e_n]$ and hashes (dictionaries) are written $\{k \Rightarrow e, \ldots\}$ where $k$ is a key (either a constant number or string). A reference $e_1[e_2]$ describes an array, a hash or a resource reference where $e_1$ itself can be a reference. When it is a resource reference, $e_1$ could be a built-in resource type. Full Puppet includes additional base types (such as floating-point numbers) and many more built-in functions that we omit here.

Statements $s$ include expressions $e$ (whose value is discarded), composite statements $s_1 \sqcup s_2$, assignments $\$x = e$, and conditionals `unless`, `if`, `case`, which are mostly standard. (Full Puppet includes an `elsif` construct that we omit from $\mu$Puppet.) Statements also

| | | | |
|---|---|---|---|
| Expression | $e$ | ::= | $i \mid w \mid \texttt{true} \mid \texttt{false} \mid \$x \mid \$::x \mid \$::a::x$ |
| | | $\mid$ | $e_1 + e_2 \mid e_1 - e_2 \mid e_1/e_2 \mid e_1 > e_2 \mid e_1 = e_2 \mid e_1 \texttt{ and } e_2 \mid e_1 \texttt{ or } e_2 \mid \texttt{!}\, e \mid \ldots$ |
| | | $\mid$ | $\{H\} \mid [e_1, \ldots, e_n] \mid e_1[e_2] \mid e \texttt{ ? } \{M\}$ |
| Array | $A$ | ::= | $\varepsilon \mid e, A$ |
| Hash | $H$ | ::= | $\varepsilon \mid k \Rightarrow e, H$ |
| Case | $c$ | ::= | $e \mid \texttt{default}$ |
| Matches | $M$ | ::= | $\varepsilon \mid c \Rightarrow e, M$ |
| Statement | $s$ | ::= | $e \mid s_1 \sqcup s_2 \mid \$x = e \mid \texttt{unless } e\ \{s\} \mid \texttt{if } e\ \{s\}\ \texttt{else}\ \{s\} \mid \texttt{case } e\ \{C\} \mid D$ |
| Cases | $C$ | ::= | $\varepsilon \mid c : \{s\} \sqcup C$ |
| Declaration | $D$ | ::= | $t\ \{e : H\} \mid u\ \{e : H\} \mid \texttt{class}\ \{a : H\} \mid \texttt{include } a$ |
| Manifest | $m$ | ::= | $s \mid m_1 \sqcup m_2 \mid \texttt{node } Q\ \{s\} \mid \texttt{define } u\ (\rho)\ \{s\} \mid \texttt{class } a\ \{s\} \mid \texttt{class } a\ (\rho)\ \{s\}$ |
| | | $\mid$ | $\texttt{class } a \texttt{ inherits } b\ \{s\} \mid \texttt{class } a\ (\rho) \texttt{ inherits } b\ \{s\}$ |
| Node spec | $Q$ | ::= | $N \mid \texttt{default} \mid (N_1, \ldots, N_k) \mid r \in \mathit{RegExp}$ |
| Parameters | $\rho$ | ::= | $\varepsilon \mid x, \rho \mid x = e, \rho$ |

**Figure 6** Abstract syntax of $\mu$Puppet

include resource declarations $t\ \{e : H\}$ for built-in resource types $t$, resource declarations $u\ \{e : H\}$ for defined resource types $u$, and class declarations $\texttt{class}\ \{a : H\}$ and $\texttt{include } a$.

Manifests $m$ can be statements $s$; composite manifests $m_1 \sqcup m_2$, class definitions $\texttt{class } a\ \{s\}$ with or without parameters $\rho$ and inheritance clauses $\texttt{inherits } b$; node definitions $\texttt{node } Q\ \{s\}$; or defined resource type definitions $\texttt{define } u\ (\rho)\ \{s\}$. Node specifications $Q$ include literal node names $N$, `default`, lists of node names, and regular expressions $r$ (which we do not model explicitly).

Sequences of statements, cases, or manifest items can be written by writing one statement after the other, separated by whitespace, and we write $\sqcup$ when necessary to emphasise that this whitespace is significant. The symbol $\varepsilon$ denotes the empty string.

## 3.2 Operational Semantics

We now define a small-step operational semantics for $\mu$Puppet. This is a considered choice: although Puppet is advertised as a declarative language, it is not *a priori* clear that manifest compilation is a terminating or even deterministic process. Using small-step semantics allows us to translate the (often) procedural descriptions of Puppet's constructs directly from the documentation.

The operational semantics relies on auxiliary notions of catalogs $v_\mathrm{C}$, scopes $\alpha$, variable environments $\sigma$, and definition environments $\kappa$ explained in more detail below. We employ three main judgements, for processing expressions, statements, and manifests:

$$\sigma, \kappa, v_\mathrm{C}, e \xrightarrow{\alpha} e' \qquad \sigma, \kappa, v_\mathrm{C}, s \xrightarrow{\alpha}_\mathsf{s} \sigma', \kappa', v'_\mathrm{C}, s' \qquad \sigma, \kappa, v_\mathrm{C}, m \xrightarrow{N}_\mathsf{m} \sigma', \kappa', v'_\mathrm{C}, m'$$

Here, $\sigma$, $\kappa$, and $v_\mathrm{C}$ are the variable environment, definition environment, and catalog beforehand, and their primed versions are the corresponding components after one compilation step. The parameter $\alpha$ for expressions and statements represents the ambient scope; the parameter $N$ for manifests is the target node name.

The main judgement is $\rightarrow_\mathsf{m}$, which takes a $\mu$Puppet manifest $m$ and a node name $N$ and compiles it to a catalog $v_\mathrm{C}$, that is, a list of resource values $v_\mathrm{R}$ for that node. Given initial variable environments $\sigma$ (representing data provided by the client) and $\kappa$ (containing any predefined classes or resource definitions), execution of manifest $m$ begins

| | | | |
|---|---|---|---|
| Catalog | $v_{\mathrm{C}}$ | ::= | $\varepsilon \mid v_{\mathrm{R}} \sqcup v_{\mathrm{C}}$ |
| Value | $v$ | ::= | $i \mid w \mid \texttt{true} \mid \texttt{false} \mid \{v_{\mathrm{H}}\} \mid [v_1, \ldots, v_n] \mid t[v]$ |
| Hash value | $v_{\mathrm{H}}$ | ::= | $\varepsilon \mid k \Rightarrow v, v_{\mathrm{H}}$ |
| Resource value | $v_{\mathrm{R}}$ | ::= | $t\,\{w : v_{\mathrm{H}}\}$ |
| Scope | $\alpha$ | ::= | $:: \mid ::a \mid ::\texttt{nd} \mid \alpha\ \texttt{def}$ |
| Statement | $s$ | ::= | $\ldots \mid \texttt{scope}\ \alpha\ s \mid \texttt{skip}$ |

**Figure 7** Auxiliary constructs: catalogs and scopes

with an empty catalog and terminates with catalog $v_{\mathrm{C}}$ when the manifest equals `skip`, i.e. $\sigma, \kappa, \varepsilon, m \xrightarrow{N}_{\mathsf{m}} \cdots \xrightarrow{N}_{\mathsf{m}} \sigma', \kappa', v_{\mathrm{C}}, \texttt{skip}$.

## 3.2.1 Auxiliary definitions: catalogs, scopes and environments

Before defining compilation formally, we first define catalogs (§3.2.1.1), the result of compiling manifests; scopes (§3.2.1.2), which explicitly represent the ambient scope used to resolve unqualified variable references; variable environments (§3.2.1.3), which store variable bindings; and definition environments (§3.2.1.4), which store class and resource definitions.

### 3.2.1.1 Catalogs

The syntax of catalogs is given in Figure 7. A *catalog* $v_{\mathrm{C}}$ is a sequence of resource values, separated by whitespace; a *resource value* $v_{\mathrm{R}} = t\,\{w : v_{\mathrm{H}}\}$ is a resource whose title is a string value and whose content is a hash value; a hash value $v_{\mathrm{H}}$ is an attribute-value sequence in which all expressions are values; and finally a *value* $v$ is either an integer literal $i$, string literal $w$, boolean literal `true` or `false`, hash $\{v_{\mathrm{H}}\}$, array $[v_1, \ldots, v_n]$ or a reference value $t[v]$. In a well-formed catalog, there is at most one resource with a given type and title; attempting to add a resource with the same type and title as one already in the catalog is an error.

### 3.2.1.2 Scopes

As discussed in Section 2, Puppet variables can be assigned in one scope and referenced in a different scope. For example, in Figure 4, the parent scope of class scope `ssh` is class scope `ssh::params`. To model this, we model scopes and parent-child relations between scopes explicitly. Scope :: represents the top scope, $::a$ is the scope of class $a$, $::\texttt{nd}$ is the active node scope, and $\alpha$ `def` is the scope of a resource definition that is executed in ambient scope $\alpha$.

The scope for defined resources takes another scope parameter $\alpha$ in order to model resource definitions that call other resource definitions. The top-level, class, and node scopes are persistent, while $\alpha$ `def` is cleared at the end of the corresponding resource definition; thus these scopes can be thought of as names for stack frames. The special statement form `scope` $\alpha$ $s$ is used internally in the semantics to model scope changes. An additional internal statement form `skip`, unrelated to scopes, represents the empty statement. Neither of these forms are allowed in Puppet manifests.

As discussed earlier, there is an ancestry relation on scopes, which governs the order in which scopes are checked when dereferencing an unqualified variable reference. We use mutually recursive auxiliary judgments $\alpha\ \mathsf{parentof}_{\kappa}\ \beta$ to indicate that $\alpha$ is the parent scope

of $\beta$ in the context of $\kappa$ and $\alpha$ $\mathsf{baseof}_\kappa$ $\beta$ to indicate that $\alpha$ is the *base* scope of $\beta$. The base scope is either ::, indicating that the scope is the top scope, or ::nd, indicating that the scope is being processed inside a node definition. We first discuss the rules for $\mathsf{parentof}_\kappa$:

$$\frac{}{:: \mathsf{parentof}_\kappa \; ::\texttt{nd}} \text{ PNode} \qquad \frac{\beta \; \mathsf{baseof}_\kappa \; \alpha \; \texttt{def}}{:: \mathsf{parentof}_\kappa \; \alpha \; \texttt{def}} \text{ PDefRes} \qquad \frac{\kappa(a) = \mathsf{DeclaredClass}(\alpha)}{\alpha \; \mathsf{parentof}_\kappa \; ::a} \text{ PClass}$$

The PNode rules says that the top-level scope is the parent scope of node scope. The PDefRes rule says that the parent scope of the defined resource type scope is its base scope. Thus, a resource definition being declared in the toplevel will have parent ::, while one being declared inside a node definition will have parent scope ::nd. The PClass rule defines the scope of the (declared) parent class $b$ to be the scope $\alpha$ that is recorded in the $\mathsf{DeclaredClass}(\alpha)$ entry. The rules for class inclusion and declaration in the next section show how the $\mathsf{DeclaredClass}(\alpha)$ entry is initialised; this also uses the $\mathsf{baseof}_\kappa$ relation. The rules defining $\mathsf{baseof}_\kappa$ are as follows:

$$\frac{}{:: \mathsf{baseof}_\kappa \; ::} \text{ BTop} \qquad \frac{}{::\texttt{nd} \; \mathsf{baseof}_\kappa \; ::\texttt{nd}} \text{ BNode} \qquad \frac{\alpha \; \mathsf{baseof}_\kappa \; \beta}{\alpha \; \mathsf{baseof}_\kappa \; \beta \; \texttt{def}} \text{ BDefRes}$$

$$\frac{\kappa(a) = \mathsf{DeclaredClass}(\beta) \qquad \alpha \; \mathsf{baseof}_\kappa \; \beta}{\alpha \; \mathsf{baseof}_\kappa \; ::a} \text{ BClass}$$

These rules determine whether the ambient scope $\alpha$ in which the class is declared is *inside* or *outside* a node declaration. The base scope of toplevel or node scope is toplevel or node scope respectively. The base scope of $\beta$ `def` is the base scope of $\beta$, while the base scope of a class scope $::a$ is the base scope of its parent scope as defined in the definition environment $\kappa$. (We will only try to obtain the base scope of a class that has already been declared.)

#### 3.2.1.3 Variable environments

During the compilation of a manifest, the values of variables are recorded in *variable environments* $\sigma$ which are then accessed when these variables are referenced in the manifest. (We call these *variable* environments, rather than plain environments, since "environment" has a specific technical meaning in Puppet; see the glossary in the appendix of the extended paper [7].) As discussed in section 2.1.3, Puppet allows variables to be referenced before being defined, whereas the variable environment is designed in the way not to allow it. A variable can only be referenced in the environment if its value has been stored. Thus the undefined variables in the manifest in Figure 2 are not legal in $\mu$Puppet.

Formally, a variable environment is defined as a partial function $\sigma : \mathit{Scope} \times \mathit{Var} \to \mathit{Value}$ which maps pairs of scopes and variables to values. The scope component indicates the scope in which the variable was assigned. We sometimes write $\sigma_\alpha(x)$ for $\sigma(\alpha, x)$. Updating a variable environment $\sigma$ with new binding $(\alpha, x)$ to $v$ is written $\sigma[(\alpha, x) : v]$, and clearing an environment (removing all bindings in scope $\alpha$) is written $\mathit{clear}(\sigma, \alpha)$.

#### 3.2.1.4 Definition environments

Some components in Puppet, like classes and defined resource types, introduce *definitions* which can be declared elsewhere. To model this, we record such definitions in *definition environments* $\kappa$. Formally, a definition environment is a partial function $\kappa : \mathit{Identifier} \to \mathit{Definition}$ mapping each identifier to a definition $D$. Evaluation of the definition only begins when a resource is declared which uses that definition.

Definitions are of the following forms:

$$\begin{array}{rcl} D & ::= & \mathsf{ClassDef}(c_{opt}, \rho, s) \mid \mathsf{DeclaredClass}(\alpha) \mid \mathsf{ResourceDef}(\rho, s) \\ c_{opt} & ::= & c \mid \bot \end{array}$$

The definition form $\mathsf{ClassDef}(c_{opt}, \rho, s)$ represents the definition of a class (before it has been declared); $c_{opt}$ is the optional name of the class's parent, $\rho$ is the list of parameters of the class (with optional default values), and $s$ is the body of the class. The definition form $\mathsf{DeclaredClass}(\alpha)$ represents a class that has been declared; $\alpha$ is the class's *parent scope* and $\rho$ and $s$ are no longer needed. In Puppet, the definition of a class can appear before or after its declaration, as we saw in the manifest in Figure 2, whereas the definition environment is designed to require that a class is defined before it is declared. Thus the inclusion of class `service1` in Figure 2 will be not evaluated in $\mu$Puppet. Moreover, a class can be declared only once in Puppet, and when it is declared its definition environment entry is changed to $\mathsf{DeclaredClass}(c_{opt})$. Finally, the definition form $\mathsf{ResourceDef}(\rho, s)$ represents the definition of a new *resource type*, where $\rho$ and $s$ are as above.

### 3.2.2 Expression evaluation

Expressions are the basic computational components of $\mu$Puppet. The rules for expression forms such as primitive operations are standard. The rules for selector expressions are also straightforward. Since variable accessibility depends on scope, the variable evaluation rules are a little more involved:

$$\frac{x \in dom(\sigma_\alpha)}{\sigma, \kappa, v_C, \$x \xrightarrow{\alpha} \sigma_\alpha(x)}\ \textsc{LVar} \qquad \frac{x \notin dom(\sigma_\alpha) \qquad \sigma, \kappa, v_C, \$x \xrightarrow{\beta} v \qquad \beta\ \mathsf{parentof}_\kappa\ \alpha}{\sigma, \kappa, v_C, \$x \xrightarrow{\alpha} v}\ \textsc{PVar}$$

$$\frac{x \in dom(\sigma_{::})}{\sigma, \kappa, v_C, \${::}x \xrightarrow{\alpha} \sigma_{::}(x)}\ \textsc{TVar} \qquad \frac{x \in dom(\sigma_{::a})}{\sigma, \kappa, v_C, \${::}a :: x \xrightarrow{\alpha} \sigma_{::a}(x)}\ \textsc{QVar}$$

The LVar looks up the value of an unqualified variable in the current scope, if present. The PVar rule handles the case of an unqualified variable that is not defined in the current scope; its value is the value of the variable in the parent scope. The TVar and QVar rules look up fully-qualified variables in top scope or class scope, respectively. (There is no qualified syntax for referencing variables in node scope from other scopes.)

$\mu$Puppet also includes array and hash expressions. An array is a list of expressions in brackets and a hash is a list of keys and their expression assignments in braces. When the expressions are values, an array or a hash is also a value. Each expression in them can be dereferenced by the array or hash followed by its index or key in brackets. The rules for constructing and evaluating arrays and hashes are straightforward, and included in the appendix of the extended paper [7].

Resource references of the form $t[v]$ are allowed as values, where $t$ is a built-in resource name and $v$ is a (string) value. Such references can be used as parameters in other resources and to express ordering relationships between resources. Resource references can be used to extend resources or override inherited resource parameters; we do not model this behaviour. A resource reference can also (as of Puppet 4) be used to access the values of the resource's parameters. This is supported in $\mu$Puppet as shown in the following example.

```
1  file {"foo.txt":
2    owner => "alice"
3  }
4  $y = "foo.txt"
```

```
5    $x = File[$y]
6    file {"bar.txt":
7      owner => $x["owner"]
8    }
```

In this example, we first declare a file resource, with an `owner` parameter `"alice"`, then we assign $y$ the filename and $\$x$ a resource reference (value) `File["foo.txt"]`. Then in defining a second file resource we refer to the `"owner"` parameter of the already-declared file resource via the reference `File["foo.txt"]`. This declaration results in a second file resource with the same owner as the first.

The rules for dereferencing arrays, hashes, and resource references are as follows:

$$\frac{\sigma, \kappa, v_C, d \xrightarrow{\alpha} d'}{\sigma, \kappa, v_C, d[e] \xrightarrow{\alpha} d'[e]}\ \textsc{DeRefExp} \qquad \frac{\sigma, \kappa, v_C, e \xrightarrow{\alpha} e'}{\sigma, \kappa, v_C, v[e] \xrightarrow{\alpha} v[e']}\ \textsc{DeRefIndex}$$

$$\frac{}{\sigma, \kappa, v_C, [v_0, \ldots, v_n, \ldots, v_m][n] \xrightarrow{\alpha} v_n}\ \textsc{DeRefArray}$$

$$\frac{k = k_n}{\sigma, \kappa, v_C, \{k_1 = v_1, \ldots, k_n = v_n, \ldots, k_m = v_m\}[k] \xrightarrow{\alpha} v_n}\ \textsc{DeRefHash}$$

$$\frac{\sigma, \kappa, v_C, e \xrightarrow{\alpha} e'}{\sigma, \kappa, v_C, t[e] \xrightarrow{\alpha} t[e']}\ \textsc{RefRes} \qquad \frac{\mathit{lookup}_C(v_C, t, w, k) = v}{\sigma, \kappa, v_C, t[w][k] \xrightarrow{\alpha} v}\ \textsc{DeRefRes}$$

In the rule DeRefExp the expression $e$ is evaluated to an array or a hash value. The rule DeRefIndex evaluates the index inside the brackets to a value. Rule DeRefArray accesses the value in an array at the index $n$ while rule DeRefHash accesses the hash value by searching its key $k$. There could be a sequence of reference indexes in a reference. As we can see, such reference is evaluated in the left-to-right order of the index list. Rule ResRef evaluates the index and in the DeRefRec rule, the function $\mathit{lookup}_C$ looks up the catalog for the value of the attribute $k$ of the resource $t[v]$.

### 3.2.3 Statement evaluation

As with expressions, some of the statement forms, such as sequential composition, conditionals (`if`, `unless`), and `case` statements have a conventional operational semantics, shown in the appendix of the extended paper [7]. An expression can occur as a statement; its value is ignored. Assignments, like variable references, are a little more complex. When storing the value of a variable in an assignment in $\sigma$, the compilation rule binds the value to $x$ in the scope $\alpha$:

$$\frac{\sigma, \kappa, v_C, e \xrightarrow{\alpha} e'}{\sigma, \kappa, v_C, \$x = e \xrightarrow{\alpha}_s \sigma, \kappa, v_C, \$x = e'}\ \textsc{AssignStep}$$

$$\frac{x \notin \mathit{dom}(\sigma_\alpha)}{\sigma, \kappa, v_C, \$x = v \xrightarrow{\alpha}_s \sigma[(\alpha, x) : v], \kappa, v_C, \texttt{skip}}\ \textsc{Assign}$$

Notice that Puppet does not allow assignment into any other scopes, only the current scope $\alpha$.

We now consider `scope` $\alpha$ $s$ statements, which are internal constructs (not part of the Puppet source language) we have introduced to track the scope that is in effect in different

parts of the manifest during execution. The following rules handle compilation inside `scope` statements and cleanup when execution inside such a statement finally terminates.

$$\frac{\alpha \in \{::, ::a, ::\texttt{nd}\} \qquad \sigma, \kappa, v_\text{C}, s \xrightarrow{\alpha}_\text{s} \sigma', \kappa', v'_\text{C}, s'}{\sigma, \kappa, v_\text{C}, \texttt{scope}\ \alpha\ s \xrightarrow{\alpha'}_\text{s} \sigma', \kappa', v'_\text{C}, \texttt{scope}\ \alpha\ s'}\ \textsc{ScopeStep}$$

$$\frac{\sigma, \kappa, v_\text{C}, s \xrightarrow{\alpha\ \texttt{def}}_\text{s} \sigma', \kappa', v'_\text{C}, s'}{\sigma, \kappa, v_\text{C}, \texttt{scope}\ (\alpha\ \texttt{def})\ s \xrightarrow{\alpha}_\text{s} \sigma', \kappa', v'_\text{C}, \texttt{scope}\ (\alpha\ \texttt{def})\ s'}\ \textsc{DefScopeStep}$$

$$\frac{\alpha \in \{::, ::a, ::\texttt{nd}\}}{\sigma, \kappa, v_\text{C}, \texttt{scope}\ \alpha\ \texttt{skip} \xrightarrow{\beta}_\text{s} \sigma, \kappa, v_\text{C}, \texttt{skip}}\ \textsc{ScopeDone}$$

$$\frac{}{\sigma, \kappa, v_\text{C}, \texttt{scope}\ (\alpha\ \texttt{def})\ \texttt{skip} \xrightarrow{\alpha}_\text{s} \mathit{clear}(\sigma, \alpha\ \texttt{def}), \kappa, v_\text{C}, \texttt{skip}}\ \textsc{DefScopeDone}$$

The ScopeStep and ScopeDef rules handle compilation inside a scope; the ambient scope $\alpha'$ is overridden and the scope parameter $\alpha$ is used instead. The ScopeDone rule handles the end of compilation inside a "persistent" scope, such as top-level, node or class scope, whose variables persist throughout execution, and the DefScopeDone rule handles the temporary scope of defined resources, whose locally-defined variables and parameters become unbound at the end of the definition. (In contrast, variables defined in toplevel, node, or class scopes remain visible throughout compilation.)

Resource declarations are compiled in a straightforward way; the title expression is evaluated, then all the expressions in attribute-value pairs in the hash component are evaluated. Once a resource is fully evaluated, it is appended to the catalog:

$$\frac{\sigma, \kappa, v_\text{C}, e : H \xrightarrow{\alpha}_\text{R} e' : H'}{\sigma, \kappa, v_\text{C}, t\ \{e : H\} \xrightarrow{\alpha}_\text{s} \sigma, \kappa, v_\text{C}, t\ \{e' : H'\}}\ \textsc{ResStep}$$

$$\frac{}{\sigma, \kappa, v_\text{C}, v_\text{R} \xrightarrow{\alpha}_\text{s} \sigma, \kappa, v_\text{C} \sqcup v_\text{R}, \texttt{skip}}\ \textsc{ResDecl}$$

Defined resource declarations look much like built-in resources:

```
1  apache::vhost {"homepages":
2    port    => 8081,
3    docroot => "/var/www-testhost",
4  }
```

When a defined resource type declaration is fully evaluated, it is expanded (much like a function call).

$$\frac{\sigma, \kappa, v_\text{C}, \{e : H\} \xrightarrow{\alpha}_\text{R} \{e' : H'\}}{\sigma, \kappa, v_\text{C}, u\ \{e : H\} \xrightarrow{\alpha}_\text{s} \sigma, \kappa, v_\text{C}, u\ \{e' : H'\}}\ \textsc{DefStep}$$

$$\frac{\kappa(u) = \mathsf{ResourceDef}(\rho, s) \qquad s' = \mathit{merge}(\rho, v_\text{H})}{\sigma, \kappa, v_\text{C}, u\ \{w : v_\text{H}\} \xrightarrow{\alpha}_\text{s} \sigma, \kappa, v_\text{C}, \texttt{scope}\ (\alpha\ \texttt{def})\ \{\$\mathit{title} = w \sqcup s' \sqcup s\}}\ \textsc{Def}$$

The *merge* function returns a statement $s'$ assigning the parameters to their default values in $\rho$ or overridden values from $v_\text{H}$. Notice that we also add the special parameter binding

$\$title = w$; this is because in Puppet, the title of a defined resource is made available in the body of the resource using the parameter $\$title$. The body of the resource definition $s$ is processed in scope $\alpha$ `def`. Class declarations take two forms: *include-like* and *resource-like declarations*.

The statement `include` $a$ is an include-like declaration of a class $a$. (Puppet includes some additional include-like declaration forms such as `contain` and `require`). Intuitively, this means that the class body is processed (declaring any ancestors and resources inside the class), and the class is marked as declared; a class can be declared at most once. The simplest case is when a class has no parent, covered by the first two rules below:

$$\frac{\kappa(a) = \mathsf{ClassDef}(\bot, \rho, s) \qquad s' = \mathit{merge}(\rho, \varepsilon) \qquad \beta\ \mathsf{baseof}_\kappa\ \alpha}{\sigma, \kappa, v_\mathrm{C}, \texttt{include}\ a \xrightarrow{\alpha}_\mathsf{s} \sigma, \kappa[a : \mathsf{DeclaredClass}(\beta)], v_\mathrm{C}, \texttt{scope}\ (::a)\ s' \sqcup s}\ \textsc{IncU}$$

$$\frac{\kappa(a) = \mathsf{DeclaredClass}(\beta)}{\sigma, \kappa, v_\mathrm{C}, \texttt{include}\ a \xrightarrow{\alpha}_\mathsf{s} \sigma, \kappa, v_\mathrm{C}, \texttt{skip}}\ \textsc{IncD}$$

$$\frac{\kappa(a) = \mathsf{ClassDef}(b, \rho, s) \qquad \kappa(b) = \mathsf{ClassDef}(c_{opt}, \rho', s')}{\sigma, \kappa, v_\mathrm{C}, \texttt{include}\ a \xrightarrow{\alpha}_\mathsf{s} \sigma, \kappa, v_\mathrm{C}, \texttt{include}\ b\ \texttt{include}\ a}\ \textsc{IncPU}$$

$$\frac{\kappa(a) = \mathsf{ClassDef}(b, \rho, s) \qquad \kappa(b) = \mathsf{DeclaredClass}(\beta) \qquad s' = \mathit{merge}(\rho, \varepsilon)}{\sigma, \kappa, v_\mathrm{C}, \texttt{include}\ a \xrightarrow{\alpha}_\mathsf{s} \sigma, \kappa[a : \mathsf{DeclaredClass}(::b)], v_\mathrm{C}, \texttt{scope}\ (::a)\ \{s' \sqcup s\}}\ \textsc{IncPD}$$

In the IncU rule, the class has not been declared yet, so we look up its body and default parameters and process the body in the appropriate scope. (We use the *merge* function again here to obtain a statement initialising all parameters which have default values.) In addition, we modify the class's entry in $\kappa$ to $\mathsf{DeclaredClass}(\beta)$, where $\beta\ \mathsf{baseof}_\kappa\ \alpha$. As described in Section 2, this aspect of Puppet scoping is dynamic: if a base class is defined outside a node definition then its parent scope is ::, whereas if it is declared during the processing of a node definition then its parent scope is ::`nd`. (As discussed below, if a class inherits from another, however, the parent scope is the scope of the parent class no matter what). If this sounds confusing, this is because it is; this is the trickiest aspect of Puppet scope that is correctly handled by $\mu$Puppet. This complexity appears to be one reason that the use of node-scoped variables is discouraged by some experts [16].

In the IncD rule, the class $a$ is already declared, so no action needs to be taken. In the IncPU rule, we include the parent class so that it (and any ancestors) will be processed first. If there is an inheritance cycle, this process loops; we have confirmed experimentally that Puppet does not check for such cycles and instead fails with a stack overflow. In the IncPD rule, the parent class is already declared, so we proceed just as in the case where there is no parent class.

The rules for *resource-like class declarations* are similar:

$$\frac{\kappa(a) = \mathsf{ClassDef}(c_{opt}, \rho, S) \qquad \sigma, \kappa, v_{\mathrm{C}}, H \xrightarrow{\alpha}_{\mathsf{H}} H'}{\sigma, \kappa, v_{\mathrm{C}}, \texttt{class}\ \{a : H\} \xrightarrow{\alpha}_{\mathsf{s}} \sigma, \kappa, v_{\mathrm{C}}, \texttt{class}\ \{a : H'\}}\ \textsc{CDecStep}$$

$$\frac{\kappa(a) = \mathsf{ClassDef}(\bot, \rho, s) \qquad s' = \mathit{merge}(\rho, v_{\mathrm{H}}) \qquad \beta\ \mathsf{baseof}_{\kappa}\ \alpha}{\sigma, \kappa, v_{\mathrm{C}}, \texttt{class}\ \{a : v_{\mathrm{H}}\} \xrightarrow{\alpha}_{\mathsf{s}} \sigma, \kappa[a : \mathsf{DeclaredClass}(\beta)], v_{\mathrm{C}}, \texttt{scope}\ (::a)\ s'_{\sqcup}s}\ \textsc{CDecU}$$

$$\frac{\kappa(a) = \mathsf{ClassDef}(b, \rho, s) \qquad \kappa(b) = \mathsf{ClassDef}(c_{opt}, \rho', s')}{\sigma, \kappa, v_{\mathrm{C}}, \texttt{class}\ \{a : v_{\mathrm{H}}\} \xrightarrow{\alpha}_{\mathsf{s}} \sigma, \kappa, v_{\mathrm{C}}, \texttt{include}\ b\ \texttt{class}\ \{a : v_{\mathrm{H}}\}}\ \textsc{CDecPU}$$

$$\frac{\kappa(a) = \mathsf{ClassDef}(b, \rho, s) \qquad \kappa(b) = \mathsf{DeclaredClass}(\beta) \qquad s' = \mathit{merge}(\rho, v_{\mathrm{H}})}{\sigma, \kappa, v_{\mathrm{C}}, \texttt{class}\ \{a : v_{\mathrm{H}}\} \xrightarrow{\alpha}_{\mathsf{s}} \sigma, \kappa[a : \mathsf{DeclaredClass}(::b)], v_{\mathrm{C}}, \texttt{scope}\ (::a)\ \{s'_{\sqcup}s\}}\ \textsc{CDecPD}$$

There are two differences. First, because resource-like class declarations provide parameters, the rule CDecStep provides for evaluation of these parameters. Second, there is no rule analogous to IncD that ignores re-declaration of an already-declared class. Instead, this is an error. (As with multiple definitions of variables and other constructs, however, we do not explicitly model errors in our rules.)

### 3.2.4 Manifest compilation

At the top level, manifests can contain statements, node definitions, resource type definitions, and class definitions. To compile statements at the top level, we use the following rule:

$$\frac{\sigma, \kappa, v_{\mathrm{C}}, s \xrightarrow{::}_{\mathsf{s}} \sigma', \kappa', v'_{\mathrm{C}}, s'}{\sigma, \kappa, v_{\mathrm{C}}, s \xrightarrow{N}_{\mathsf{m}} \sigma', \kappa', v'_{\mathrm{C}}, s'}\ \textsc{TopScope}$$

The main point of interest here is that we change from the manifest judgement (with the node name parameter $N$) to the statement judgement (with toplevel scope parameter ::). The node name parameter is not needed for processing statements, and we (initially) process statements in the toplevel scope. Of course, the statement $s$ may well itself be a `scope` statement which immediately changes the scope.

A manifest in Puppet can configure all the machines (nodes) in a system. A node definition describes the configuration of one node (or type of nodes) in the system. The node declaration includes a specifier $Q$ used to match against the node's hostname. We abstract this matching process as a function $\mathsf{nodeMatch}(N, Q)$ that checks if the name $N$ of the requesting computer matches the specifier $Q$. If so (NodeMatch) we will compile the statement body of $N$. Otherwise (NodeNoMatch) we will skip this definition and process the rest of the manifest.

$$\frac{\mathsf{nodeMatch}(N, Q)}{\sigma, \kappa, v_{\mathrm{C}}, \texttt{node}\ Q\ \{s\} \xrightarrow{N}_{\mathsf{m}} \sigma, \kappa, v_{\mathrm{C}}, \texttt{scope}\ (::\texttt{nd})\ s}\ \textsc{NodeMatch}$$

$$\frac{\neg\mathsf{nodeMatch}(N, Q)}{\sigma, \kappa, v_{\mathrm{C}}, \texttt{node}\ Q\ \{s\} \xrightarrow{N}_{\mathsf{m}} \sigma, \kappa, v_{\mathrm{C}}, \texttt{skip}}\ \textsc{NodeNoMatch}$$

Resource type definitions in Puppet are used to design new, high-level resource types, possibly by declaring other built-in resource types, defined resource types, or classes. Such a

definition includes Puppet code to be executed when the a resource of the defined type is declared. Defined resource types can be declared multiple times with different parameters, so resource type definitions are loosely analogous to procedure calls. The following is an example of a defined resource type:

```
define apache::vhost (Integer $port) {
  include apache
  file { "host":
    content => template('apache/vhost-default.conf.erb'),
    owner   => 'www'
  }
}
```

The compilation rule for defining a defined resource type is:

$$\frac{u \notin dom(\kappa)}{\sigma, \kappa, v_C, \texttt{define}\ u\ (\rho)\ \{s\} \xrightarrow{N}_{\mathsf{m}} \sigma, \kappa[u : \mathsf{ResourceDef}(\rho, s)], v_C, \texttt{skip}}\ \textsc{RDef}$$

The definition environment is updated to map $u$ to $\mathsf{ResourceDef}(\rho, s)$ containing the parameters and statements in the definition of $u$. The manifest then becomes `skip`.

A class definition is used for specifying a particular service that could include a set of resources and other statements. Classes are defined at the top level and are declared as part of statements, as described earlier. Classes can be parameterised; the parameters are passed in at declaration time using the resource-like declaration syntax. The parameters can be referenced as variables in the class body. A class can also inherit directly from one other class. The following rules handle the four possible cases:

$$\frac{a \notin dom(\kappa)}{\sigma, \kappa, v_C, \texttt{class}\ a\ \{s\} \xrightarrow{N}_{\mathsf{m}} \sigma, \kappa[a : \mathsf{ClassDef}(\bot, \varepsilon, s)], v_C, \texttt{skip}}\ \textsc{CDef}$$

$$\frac{a \notin dom(\kappa)}{\sigma, \kappa, v_C, \texttt{class}\ a\ \texttt{inherits}\ b\ \{s\} \xrightarrow{N}_{\mathsf{m}} \sigma, \kappa[a : \mathsf{ClassDef}(b, \varepsilon, s)], v_C, \texttt{skip}}\ \textsc{CDefI}$$

$$\frac{a \notin dom(\kappa)}{\sigma, \kappa, v_C, \texttt{class}\ a\ (\rho)\ \{s\} \xrightarrow{N}_{\mathsf{m}} \sigma, \kappa[a : \mathsf{ClassDef}(\bot, \rho, s)], v_C, \texttt{skip}}\ \textsc{CDefP}$$

$$\frac{a \notin dom(\kappa)}{\sigma, \kappa, v_C, \texttt{class}\ a\ (\rho)\ \texttt{inherits}\ b\ \{s\} \xrightarrow{N}_{\mathsf{m}} \sigma, \kappa[a : \mathsf{ClassDef}(b, \rho, s)], v_C, \texttt{skip}}\ \textsc{CDefPI}$$

In the simples case (CDef) we add the class definition to the definition environment with no parent and no parameters. The other three rules handle the cases with inheritance, with parameters, or with both.

## 4 Metatheory

Because Puppet has not been designed with formal properties in mind, there is relatively little we can say formally about the "correctness" of $\mu$Puppet. Instead, the main measure of correctness is the degree to which $\mu$Puppet agrees with the behaviour of the main Puppet implementation, which is the topic of the next section. Here, we summarise two properties of $\mu$Puppet that guided our design of the rules, and provide some justification for the claim that $\mu$Puppet is 'declarative'. First, evaluation is deterministic: a given manifest can evaluate in at most one way.

▶ **Theorem 1** (Determinism). *All of the evaluation relations of μPuppet are deterministic:*

- *If* $\sigma, \kappa, v_C, e \xrightarrow{\alpha} e'$ *and* $\sigma, \kappa, v_C, e \xrightarrow{\alpha} e''$ *then* $e' = e''$.
- *If* $\sigma, \kappa, v_C, s \xrightarrow{\alpha}_s \sigma', \kappa', v'_C, s'$ *and* $\sigma, \kappa, v_C, s \xrightarrow{\alpha}_s \sigma'', \kappa'', v''_C, s''$ *then* $\sigma' = \sigma''$, $\kappa' = \kappa''$, $v'_C = v''_C$ *and* $s' = s''$.
- *If* $\sigma, \kappa, v_C, s \xrightarrow{N}_m \sigma', \kappa', v'_C, m'$ *and* $\sigma, \kappa, v_C, m \xrightarrow{N}_m \sigma'', \kappa'', v''_C, m''$ *then* $\sigma' = \sigma''$, $\kappa' = \kappa''$, $v'_C = v''_C$ *and* $m' = m''$.

**Proof.** Straightforward by induction on derivations. ◀

Second, in μPuppet, evaluation is monotonic in the sense that:

- Once a variable binding is defined in $\sigma$, its value never changes, and it remains bound until the end of the scope in which it was bound.
- Once a class or resource definition is defined in $\kappa$, its definition never changes, except that a class's definition may change from $\mathsf{ClassDef}(c_{opt}, \rho, s)$ to $\mathsf{DeclaredClass}(\beta)$.
- Once a resource is declared in $v_C$, its title, properties and values never change.

We can formalise this as follows.

▶ **Definition 2.** We define orderings $\sqsubseteq$ on variable environments, definition environments and catalogs as follows:

- $\sigma \sqsubseteq \sigma'$ when $x \in dom(\sigma_\alpha)$ implies that either $\sigma_\alpha(x) = \sigma'_\alpha(x)$ or $\alpha = \beta$ `def` for some $\beta$ and $x \notin dom(\sigma'_\alpha)$.
- $\kappa \sqsubseteq \kappa'$ when $a \in dom(\kappa)$ implies either $\kappa(a) = \kappa'(a)$ or $\kappa(a) = \mathsf{ClassDef}(c_{opt}, \rho, s)$ and $\kappa'(a) = \mathsf{DeclaredClass}(\beta)$.
- $v_C \sqsubseteq v'_C$ when there exists $v''_C$ such that $v_C \sqcup v''_C = v'_C$.
- $(\sigma, \kappa, v_C) \sqsubseteq (\sigma', \kappa', v'_C)$ when $\sigma \sqsubseteq \sigma'$, $\kappa \sqsubseteq \kappa'$ and $v_C \sqsubseteq v'_C$.

▶ **Theorem 3** (Monotonicity). *The statement and manifest evaluation relations of μPuppet are monotonic in* $\sigma, \kappa, v_C$*:*

- *If* $\sigma, \kappa, v_C, s \xrightarrow{\alpha}_s \sigma', \kappa', v'_C, s'$ *then* $(\sigma, \kappa, v_C) \sqsubseteq (\sigma', \kappa', v'_C)$.
- *If* $\sigma, \kappa, v_C, s \xrightarrow{N}_m \sigma', \kappa', v'_C, m'$ *then* $(\sigma, \kappa, v_C) \sqsubseteq (\sigma', \kappa', v'_C)$.

**Proof.** Straightforward by induction. The only interesting cases are the rules in which $\sigma$, $\kappa$ or $v_C$ change; in each case the conclusion is immediate. ◀

These properties are not especially surprising or difficult to prove; nevertheless, they provide some justification for calling μPuppet a 'declarative' language. However, μPuppet does not satisfy some other desirable properties. For example, as we have seen, the order in which variable definitions or resource or class declarations appear can affect the final result. Likewise, there is no notion of 'well-formedness' that guarantees that a μPuppet program terminates successfully: compilation may diverge or encounter a run-time error. Furthermore, full Puppet does not satisfy monotonicity, because of other non-declarative features that we have chosen not to model. Further work is needed to identify and prove desirable properties of the full Puppet language, and identify subsets of (or modifications to) Puppet that make such properties valid.

## 5 Implementation and Evaluation

We implemented a prototype parser and evaluator μPuppet in Haskell (GHC 8.0.1). The parser accepts source syntax for features of μPuppet as described in the Puppet documentation and produces abstract syntax trees as described in Section 3.2. The evaluator implements μPuppet compilation based on the rules shown in the appendix of the extended paper [7].

| Feature | #Tests | #Pass | #Unsupported |
|---|---|---|---|
| Statements | 11 | 11 | 0 |
| Assignment | 2 | 2 | 0 |
| Case | 1 | 1 | 0 |
| If | 4 | 4 | 0 |
| Unless | 4 | 4 | 0 |
| Resources | 18 | 11 | 7 |
| Basics | 2 | 2 | 0 |
| Variables | 3 | 3 | 0 |
| User defined resource types | 5 | 5 | 0 |
| *Virtual resources* | 1 | 0 | 1 |
| *Default values* | 1 | 0 | 1 |
| *Resource extension* | 4 | 0 | 4 |
| *Ordering Constraints* | 2 | 1 | 1 |
| Classes | 32 | 22 | 10 |
| Basics | 4 | 4 | 0 |
| Inheritance | 3 | 3 | 0 |
| Scope | 2 | 2 | 0 |
| Variables & classes | 6 | 6 | 0 |
| Class Parameters | 6 | 6 | 0 |
| *Overriding* | 5 | 0 | 5 |
| *Nesting and redefinition* | 6 | 1 | 5 |
| Nodes | 8 | 8 | 0 |
| *Resource Collectors* | 9 | 0 | 9 |
| *Basics* | 1 | 0 | 1 |
| *Collectors, references & variables* | 3 | 0 | 3 |
| *Application order* | 5 | 0 | 5 |

**Table 1** Summary of test cases. Features in *italics* are not supported in $\mu$Puppet.

The implementation constitutes roughly 1300 lines of Haskell code. The evaluator itself is roughly 400 lines of code, most of which are line-by-line translations of the inference rules.

We also implemented a test framework that creates an Ubuntu 16.04.1 (x86_64) virtual machine with Puppet installed, and scripts which run each example using both $\mu$Puppet and Puppet and compare the resulting messages and catalog output.

## 5.1 Test cases and results

During our early investigations with Puppet, we constructed a test set of 52 manifests that illustrate Puppet's more unusual features, including resources, classes, inheritance, and resource type definitions. The tests include successful examples (where Puppet produces a catalog) and failing examples (where Puppet fails with an error); we found both kinds of tests valuable for understanding what is possible in cases where the documentation was unspecific.

We used these test cases to guide the design of $\mu$Puppet, and developed 16 additional test cases along the way to test corner cases or clarify behaviour that our rules did not originally capture correctly. We developed further tests during debugging and to check the behaviour of Puppet's (relatively) standard features, such as conditionals and case statements, arrays,

and hashes. We did not encounter any surprises there so we do not present these results in detail.

We summarise the test cases and their results in Table 1. The "Feature" column describes the classification of features present in our test set. The "#Tests" and "#Pass" columns show the number of tests in each category and the number of them that pass. A test that is intended to succeed passes if both Puppet and $\mu$Puppet succeed and produce the same catalog (up to reordering of resources); a test that is intended to fail passes if both Puppet and $\mu$Puppet fail. The "#Unsupported" column shows the number of test cases that involve features that $\mu$Puppet does not handle. All of the tests either pass or use features that are not supported by $\mu$Puppet. Features that $\mu$Puppet (by design) does not support are italicised.

All of the examples listed in the above table are included in the supplementary material, together with the resulting catalogs and error messages provided by Puppet.

## 5.2 Other Puppet examples

A natural source of test cases is Puppet's own test suite or, more generally, other Puppet examples in public repositories. Puppet does have a test suite, but it is mostly written in Ruby to test internal functionality. We could find only 43 Puppet language tests in the Puppet repository on GitHub[3]. These tests appear to be aimed at testing parsing and lexing functionality; they are not accompanied by descriptions of the desired catalog result. Some of the tests also appear to be out of date: five fail in Puppet 4.8. Of the remaining test cases that Puppet 4.8 can run, 20 run correctly in $\mu$Puppet (with minor modifications) while 18 use features not yet implemented in $\mu$Puppet.

We also considered harvesting realistic Puppet configurations from other public repositories; however, this is not straightforward since real configurations typically include confidential or security-critical parameters so are not publicly available. An alternative would be to harvest Puppet modules from publicly available sources such as PuppetForge, which often include test manifests to show typical usage. However, these test cases usually do not come with sample results; they are mainly intended for illustration.

We examined the top 10 Puppet modules (apache, ant, concat, firewall, java, mysql, ntp, postgresql, puppetdb, and stdlib) on the official PuppetForge module site and searched for keywords and other symbols in the source code to estimate the number of uses of Puppet features such as classes, inheritance, definitions, resource collectors/virtual resources, and ordering constraints. Classes occurred in almost all modules, with over 200 uses overall, and over 50 uses of inheritance. Resource type definitions were less frequent, with only around 40 uses, while uses of resource collectors and virtual resources were rare: there were only 10 uses overall, distributed among 5 packages. Ordering constraints were widely used, with over 90 occurrences in 8 packages. Due to the widespread use of ordering constraints, as well as other issues such as the lack of support for general strings and string interpolation in $\mu$Puppet, we were not able to run $\mu$Puppet on these Puppet modules. This investigation suggests that to develop tools or analyses for real Puppet modules based on $\mu$Puppet will require both conceptual steps (modelling ordering constraints and non-declarative features such as resource collectors) as well as engineering effort (e.g. to handle Puppet's full, idiosyncratic string interpolation syntax).

[3] https://github.com/puppetlabs/puppet/tree/master/spec/fixtures/unit/parser/lexer

## 5.3 Unsupported features

Our formalisation handles some but not all of the distinctive features of Puppet. As mentioned in the introduction, we chose to focus effort on the well-established features of Puppet that appear closest to its declarative aspirations. In this section we discuss the features we chose not to support and how they might be supported in the future, in increasing order of complexity.

**String interpolation**. Puppet supports a rich set of string operations including string interpolation (i.e. variables and other expression forms embedded in strings). For example, writing `"Hello ${planet['earth']}!"` produces `"Hello world!"` if variable `planet` is a hash whose `'earth'` key is bound to `'World'`. String interpolation is not conceptually difficult but it is widely used and desugaring it correctly to plain string append operations is an engineering challenge.

**Dynamic data types**. Puppet 4 also supports type annotations, which are checked dynamically and can be used for automatic validation of parameters. For example, writing `Integer $x = 5` in a parameter list says that $x$ is required to be an integer and its default value is 5. Types can also express constraints on the allowed values: for example, `5 =~ Integer[1,10]` is a valid expression that evaluates to `true` because 5 is an integer between 1 and 10. Data types are themselves values and there is a type `Type` of data types.

**Undefined values and strict mode**. By default, Puppet treats an undefined variable as having a special "undefined value" `undef`. Puppet provides a "strict" mode that treats an attempt to dereference an undefined variable as an error. We have focused on modelling strict semantics, so our rules get stuck if an attempt is made to dereference an undefined variable; handling explicit undefined values seems straightforward, by changing the definitions of lookup and related operations to return `undef` instead of failing.

**Functions, iteration and lambdas**. As of version 4, Puppet allows function definitions to be written in Puppet and also includes support for iteration functions (`each`, `slice`, `filter`, `map`, `reduce`, `with`) which take lambda blocks as arguments. The latter can only be used as function arguments, and cannot be assigned to variables, so Puppet does not yet have true first-class functions. We do see no immediate obstacle to handling these features, using standard techniques.

**Nested constructs and multiple definitions**. We chose to consider only top-level definitions of classes and defined resources, but Puppet allows nesting of these constructs, which also makes it possible for classes to be defined more than once. For example:

```
1  class a {
2    $x1 = "a"
3    class b {
4      $y1 = "b"
5    }
6  }
7
8  class a::b {
9    $y2 = "ab"
10 }
11 include a
12 include a::b
```

Surprisingly, *both* line 4 and line 9 are executed (in unspecified order) when `a::b` is declared, so both `$::a::b::y1` and `$::a::b::y2` are in scope at the end. Our impression is that it would be better to simply reject Puppet manifests that employ either nested classes or multiple definitions, since nesting of class and resource definitions is explicitly discouraged by the Puppet documentation.

**Dynamically-scoped resource defaults**. Puppet also allows setting resource defaults. For example one can write (using the capitalised resource type `File`):

```
File { owner => "alice" }
```

to indicate that the default owner of all files is `alice`. Defaults can be declared in classes, but unlike variables, resourced defaults are dynamically scoped; for this reason, the documentation and some authors both recommend using resource defaults sparingly. Puppet 4 provides an alternative way to specify defaults as part of the resource declaration.

**Resource extension and overriding**. In Puppet, attributes can be added to a resource which has been previously defined by using a *reference* to the resource, or removed by setting them to `undef`.

```
class main {
  file { "file": owner => "alice" }
  File["file"] { mode => "0755" }
}
```

However, it is an error to attempt to change the value of an already-defined resource, unless the updating operation is performed in a *subclass* of the class in which the resource was originally declared. For example:

```
class main::parent {
  file { "file":
    owner => "bob",
    source => "the source"
  }
}
class main inherits main::parent {
  File["file"] {
    owner => "alice",
    source => undef
  }
}
```

This illustrates that code in the derived class is given special permission to override any resource attributes that were set in the base class. Handling this behaviour seems to require (at least) tracking the classes in which resources are declared.

**Resource collectors and virtual resources**. *Resource collectors* allow for selecting, and also updating, groups of resources specified via predicates. For example, the following code declares a resource and then immediately uses the collector `File <|title == file"|>"` to modify its parameters.

```
class main {
        file { "file": owner => "alice" }
        File <| title == "file" |> {
                owner => "bob",
                group => "the group",
        }
}
```

Updates based on resource collectors are unrestricted, and the scope of the modification is also unrestricted: so for example the resource collector `File<|owner='root'|>` will select all files owned by root, and potentially update their parameters in arbitrary ways. The Puppet documentation recommends using resource collectors only in idiomatic ways, e.g. using the title of a known resource as part of the predicate. Puppet also supports *virtual resources*, that is, resources with parameter values that are not added to the catalog until declared or referenced elsewhere. Virtual resources allow a resource to be declared in one place without the resource being included in the catalog. The resource can then be *realised* in one or more other places to include it in the catalog. Notice that you can realise virtual resources before declaring them:

```
class main {
        realize User["alice"]
```

```
3           @user { "alice": uid => 100 }
4           @user { "bob": uid => 101 }
5           realize User["alice"]
6   }
```

As Shambaugh et al. [17] observe, these features can have global side-effects and make separate compilation impossible; the Puppet documentation also recommends avoiding them if possible. We have not attempted to model these features formally, and doing so appears to be a challenging future direction.

**Ordering constraints**. By default, Puppet does not guarantee to process the resources in the catalog in a fixed order. To provide finer-grained (and arguably more declarative) control over ordering, Puppet provides several features: special *metaparameters* such as `ensure`, `require`, `notify`, and `subscribe`, *chaining arrows* `->` and `~>` that declare dependencies among resources, and the *require function* that includes a class and creates dependencies on its resources. From the point of view of our semantics, all of these amount to ways to define dependency edges among resources, making the catalog into a *resource graph*. Puppet represents the chaining arrow dependencies using metaparameters, so we believe this behaviour can be handled using techniques similar to those for resource parameter overrides or resource collectors. The rules for translating the different ordering constraints to resource graph edges can be expressed using Datalog rules [17] and this approach may be adaptable to our semantics too.

## 6 Related work

Other declarative configuration frameworks include LCFG [2], a configuration management system for Unix, and SmartFrog [9], a configuration language developed by HP Labs. Of these, only SmartFrog has been formally specified; Herry and Anderson [4] propose a formal semantics and identify some complications, including potential termination problems exhibited by the SmartFrog interpreter. Their semantics is presented in a denotational style, in contrast to the small-step operational semantics presented here for Puppet. Other systems, such as Ponder [5], adopt an operational approach to policies for distributed systems.

Beyond this, there are relatively few formal studies of configuration languages, and we are aware of only two papers on Puppet specifically. Vanbrabant et al. [20] propose an access control technique for an early version of Puppet based on checking whether the changes to the catalog resulting from a change to the manifest are allowed by a policy. Catalogs are represented as XML files and allowed changes are described using path expressions. Shambaugh et al. [17] present a configuration verification tool for Puppet called Rehearsal. Their tool is concerned primarily with the "realisation" stage of a Puppet configuration, and focuses on the problem of determinacy analysis: that is, determining whether a proposed reconfiguration leads to a unique result state. Shambaugh et al. consider a subset of Puppet as a source language, including resources, defined resources, and dependencies. However, some important subtleties of the semantics were not investigated. Compilation of definitions and ordering constraints was described at a high level but not formalised; classes and inheritance were not mentioned, although their implementation handles simple cases of these constructs. Our work complements Rehearsal: Rehearsal analyses the determinacy of the realisation stage, while our work improves understanding of the compilation stage.

The present work continues a line of recent efforts to study the semantics of programming and scripting languages "in the wild". There have been efforts to define semantics for JavaScript [12, 10], R [14], PHP [6], and Python [15]. Work on formal techniques for Ruby [19] may be especially relevant to Puppet: Puppet is implemented in Ruby, and plugins

can be written in Ruby, so modelling the behaviour of Puppet as a whole may require modelling both the Puppet configuration language and the Ruby code used to implement plugins, as well as other tools such as Hiera[4] that are an increasingly important component of the Puppet toolchain. However, Puppet itself differs significantly from Ruby, and Puppet "classes" in particular bear little relation to classes in Ruby or other object-oriented languages.

## 7 Conclusions

Rigorous foundations for configuration frameworks are needed to improve the reliability of configurations for critical systems. Puppet is a popular configuration framework, and is already being used in safety-critical domains such as air traffic control.[5]

Even if each individual component of such a system is formally verified, misconfiguration errors can still lead to failures or vulnerabilities, and the use of these tools at scale means that the consequences of failure are also potentially large-scale. The main contribution of this paper is an operational semantics for a subset of Puppet, called $\mu$Puppet, that covers the distinctive features of Puppet that are used in most Puppet configurations, including resource, node, class, and defined resource constructs. Our rules also model Puppet's idiosyncratic treatment of classes, scope, and inheritance, including the dynamic treatment of node scope.

We presented some simple metatheoretic properties that justify our characterisation of $\mu$Puppet as a 'declarative' subset of Puppet, and we compared $\mu$Puppet with the Puppet 4.8 implementation on a number of examples. We also identified idiosyncrasies concerning evaluation order and scope where our initial approach differed from Puppet's actual behaviour. Because Puppet is a work in progress, we hope that these observations will contribute to the evolution and improvement of the Puppet language. In future work, we plan to investigate more advanced features of Puppet and develop semantics-based analysis and debugging techniques; two natural directions for future work are investigating Puppet's recently-added type system, and developing *provenance* techniques that can help explain where a catalog value or resource came from, why it was declared, or why manifest compilation failed [3].

### Acknowledgments

Fu was supported by a Microsoft Research PhD studentship. Perera and Cheney were supported by the Air Force Office of Scientific Research, Air Force Material Command, USAF, under grant number FA8655-13-1-3006. The U.S. Government and University of Edinburgh are authorised to reproduce and distribute reprints for their purposes notwithstanding any copyright notation thereon. Perera was also supported by UK EPSRC project EP/K034413/1. We also gratefully acknowledge Arjun Guha for comments on an early version of this paper and Henrik Lindberg for discussions about Puppet's semantics and tests.

[4] `https://docs.puppet.com/hiera/`.
[5] `https://archive.fosdem.org/2011/schedule/event/puppetairtraffic.html`.

## References


1 Puppet 4.8 reference manual, 2016. https://docs.puppet.com/puppet/4.8/index.html.

2 Paul Anderson. *LCFG: a Practical Tool for System Configuration*, volume 17 of *Short Topics in System Administration*. Usenix Association, 2008.

3 Paul Anderson and James Cheney. Toward provenance-based security for configuration languages. In *TaPP*. USENIX, 2012. Online proceedings: http://www.usenix.org/system/files/conference/tapp12/tapp12-final15.pdf.

4 Paul Anderson and Herry Herry. A formal semantics for the SmartFrog configuration language. *J. Network Syst. Manage.*, 24(2):309–345, 2016. URL: http://dx.doi.org/10.1007/s10922-015-9351-y, doi:10.1007/s10922-015-9351-y.

5 Nicodemos Damianou. *A policy framework for management of distributed systems*. PhD thesis, Imperial College, 2002.

6 Daniele Filaretti and Sergio Maffeis. An executable formal semantics of PHP. In *ECOOP*, pages 567–592, 2014. URL: http://dx.doi.org/10.1007/978-3-662-44202-9_23, doi:10.1007/978-3-662-44202-9_23.

7 Weili Fu, Roly Perera, Paul Anderson, and James Cheney. $\mu$Puppet: A declarative subset of the Puppet configuration language. *ArXiv e-prints*, August 2016. arXiv:1608.04999.

8 Jeff Geerling. *Ansible for DevOps: Server and configuration management for humans*. Midwestern Mac, LLC, 2015.

9 Patrick Goldsack, Julio Guijarro, Steve Loughran, Alistair Coles, Andrew Farrell, Antonio Lain, Paul Murray, and Peter Toft. The SmartFrog configuration management framework. *SIGOPS Oper. Syst. Rev.*, 43(1):16–25, January 2009. URL: http://doi.acm.org/10.1145/1496909.1496915, doi:10.1145/1496909.1496915.

10 Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of JavaScript. In *ECOOP*, pages 126–150, Berlin, Heidelberg, 2010. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=1883978.1883988.

11 Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. What bugs live in the cloud? a study of 3000+ issues in cloud systems. In *SOCC*, pages 7:1–7:14, New York, NY, USA, 2014. ACM. URL: http://doi.acm.org/10.1145/2670979.2670986, doi:10.1145/2670979.2670986.

12 Sergio Maffeis, John C. Mitchell, and Ankur Taly. An operational semantics for JavaScript. In *APLAS*, pages 307–325, Berlin, Heidelberg, 2008. Springer-Verlag. URL: http://dx.doi.org/10.1007/978-3-540-89330-1_22, doi:10.1007/978-3-540-89330-1_22.

13 Matthias Marschall. *Chef Infrastructure Automation Cookbook*. Packt Publishing, 2013.

14 Floréal Morandat, Brandon Hill, Leo Osvald, and Jan Vitek. Evaluating the design of the R language - objects and functions for data analysis. In *ECOOP*, pages 104–131, 2012. URL: http://dx.doi.org/10.1007/978-3-642-31057-7_6, doi:10.1007/978-3-642-31057-7_6.

15 Joe Gibbs Politz, Alejandro Martinez, Matthew Milano, Sumner Warren, Daniel Patterson, Junsong Li, Anand Chitipothu, and Shriram Krishnamurthi. Python: The full monty. In *OOPSLA*, pages 217–232, New York, NY, USA, 2013. ACM. URL: http://doi.acm.org/10.1145/2509136.2509536, doi:10.1145/2509136.2509536.

16 Jo Rhett. *Learning Puppet 4: A guide to configuration management and automation*. O'Reilly Media, 2016.

17 Rian Shambaugh, Aaron Weiss, and Arjun Guha. Rehearsal: a configuration verification tool for Puppet. In *PLDI*, pages 416–430, 2016. URL: http://doi.acm.org/10.1145/2908080.2908083, doi:10.1145/2908080.2908083.

18 James Turnbull. *Pulling Strings with Puppet: Configuration Management Made Easy*. Apress, September 2008.

19 Katsuhiro Ueno, Yutaka Fukasawa, Akimasa Morihata, and Atsushi Ohori. The essence of Ruby. In *APLAS*, pages 78–98, 2014. URL: http://dx.doi.org/10.1007/978-3-319-12736-1_5, doi:10.1007/978-3-319-12736-1_5.

20 Bart Vanbrabant, Joris Peeraer, and Wouter Joosen. Fine-grained access control for the Puppet configuration language. In *LISA*, December 2011. URL: https://lirias.kuleuven.be/handle/123456789/316070.

21 Tianyin Xu and Yuanyuan Zhou. Systems approaches to tackling configuration errors: A survey. *ACM Comput. Surv.*, 47(4):70:1–70:41, July 2015. URL: http://doi.acm.org/10.1145/2791577, doi:10.1145/2791577.

22 Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram, and Shankar Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 159–172, New York, NY, USA, 2011. ACM. URL: http://doi.acm.org/10.1145/2043556.2043572, doi:10.1145/2043556.2043572.

23 Diego Zamboni. *Learning CFEngine 3: Automated system administration for sites of any size*. O'Reilly Media, 2012.